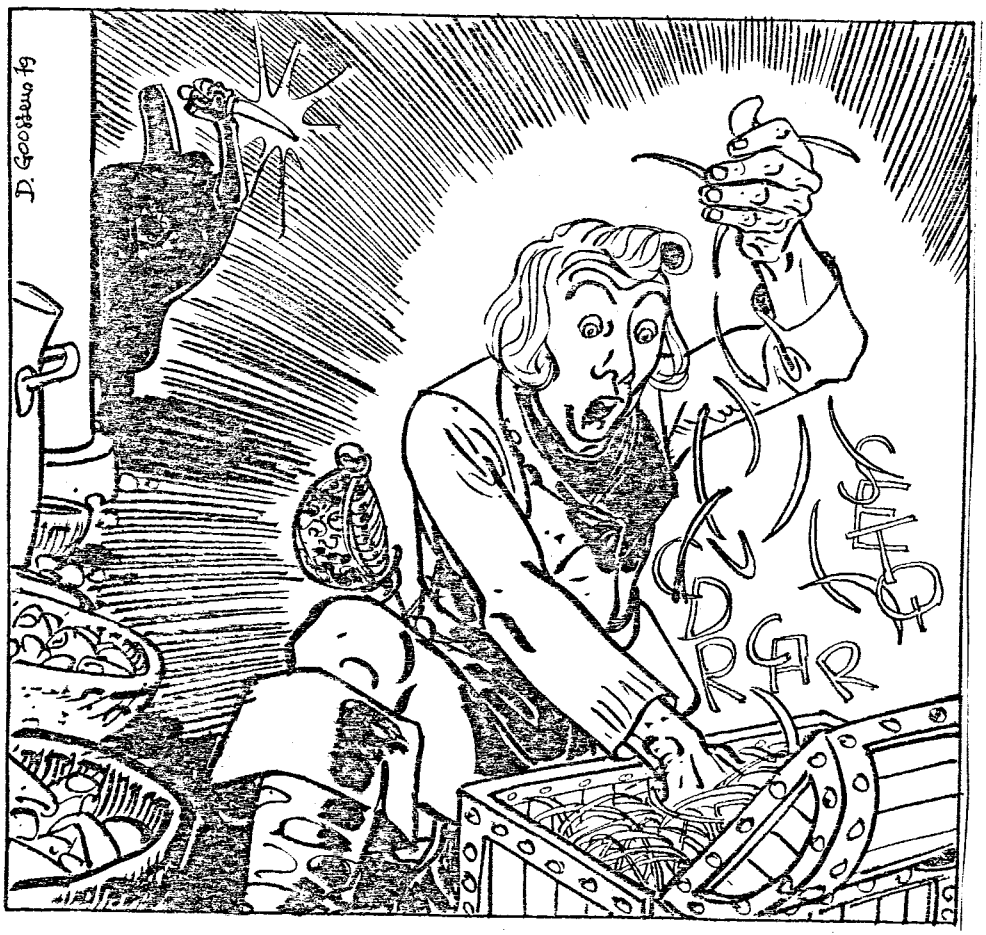
(LISP BULLETIN)



#3

December 1979

Editors : **P. GREUSSAY**
Dépt. Informatique, Université Paris-8-Vincennes
Route de la Tourelle, Paris 75012, FRANCE

J. LAUBSCH
Institut fur Informatik, Universitat Stuttgart
Azenbergstr. 12, 7000 Stuttgart 1, BRD

Table of Contents

from the editors

Well, here is another **(LISP BULLETIN)**. This is the third issue (which explains the #3 on the cover). Our humble world-famous bulletin gets better each issue : the cover design is again by Daniel GOOSSENS and the technical articles are of the highest standards of excellence. Reaction to #2 has been quite promising.

(LISP BULLETIN) is one of the few things which does not cost more than it did last year: it's still free, but don't expect ***that*** to last much longer.

Since **LISP** is the best language in the world, everyone will want to know about the forthcoming **LISP** Conference, an unprecedented marvel full of wonder and delight.

As usual, we welcome contributions. The following topics are particularly encouraged

- small technical papers
- useful functions
- comments
- puzzles
- announcements for further new **LISP** systems
- book and paper reviews
- original engravings by M. C. Escher

As space permits, we are also interested in publishing ***programs***, as well as smaller useful functions. Remember that we are not retyping communications that we publish. So the program listings should be printed on standard 8 1/2 * 11 inch, white paper.

CALL FOR PAPERS
1980 LISP Conference

The 1980 LISP Conference hosted by Stanford University, will be held on the Stanford campus, August 24-27, 1980. A proceedings will be published.

PURPOSE. Many areas of contemporary computer science have their spiritual roots in developments related to LISP. These areas include machine architecture, systems design, programming methodology and technology, and a theory of computation. The call for papers reflects this breadth.

TOPICS. The following topics are typical, but not exclusive:

Languages and Theory. Applicative languages, Object-oriented languages, Proving correctness of LISP programs, Mathematics and formal semantics of LISP-like languages.

Programming Aspects. Programming tools and environments for LISP-like languages, Applications of these ideas to other languages.

Architecture. The design and implementation of LISP hardware, Adaptation of existing machines, Specially designed LISP machines.

Applications. Non-traditional applications of LISP. This area, of course, is not easily categorized.

PAPER SUBMITTAL. Authors are requested to send four copies of a full draft paper not exceeding 4500 words, and a one-page abstract, by March 14, 1980 to the Conference Head.

The abstract should provide sufficient detail to allow the committee to apply uniform criteria for acceptance. Appropriate references and comparison to extant work should be included. The papers will be "blind refereed". Traces of authorship should not appear within the body of the paper; this information should appear *only* in a cover letter to the Conference Head.

Authors will be notified of acceptance or rejection by May 16, 1980. For inclusion in the proceedings, final papers are due by June 27, 1980.

PROGRAM COMMITTEE. The committee consists of: John R. Allen, Bruce Anderson, Richard Fateman, Dan Friedman, Eiichi Goto, Patrick Greussay, Tony Hearn, Carl Hewitt, Alan Kay, Peter Landin, Joachim Laubsch, John McCarthy, Gianfranco Prini, Erik Sandewall, Carolyn Talcott, and David Wise.

IMPORTANT ADDRESSES.

Conference Head is:
John R. Allen
Stanford Artificial Intelligence Lab
Stanford University
Stanford California 94305
(415)497-4971

In Charge of Local Arrangements is:
Dr. Ruth E. Davis
Department of EECS
University of Santa Clara
Santa Clara, California 95053
(408)984-4358

MEETING FORMAT. Besides the formal sessions, we expect to have several demonstrations, including LISP machines.

Evening sessions may be established, and informal workshops will be encouraged.

PANEL DISCUSSION. Tuesday evening, August 26, 1980, there will be a panel discussion on the topic "What is LISP?". Even informal conversations will elicit several divergent if not contradictory views of LISP; a organized effort should prove even more illuminating.

(HOFSTADTER)

Write a **LISP** function to generate the following set of integers

(1 3 7 12 18 26 35 45 56 ...

(HOFSTADTER)

This is the function **g**. What is she doing?

```
(DE g (n)
    (IF (ZEROP n) 0
        (- n (g (g (SUB1 n))))))
```

And what about the function **q**?

```
(DE q (n)(COND
     ((= n 1) 1)
     ((= n 2) 1)
     (T (+ (q (- n (q (SUB1 n))))
           (q (- n (q (- n 2))))))))
```

(BRATLEY, MILO)

This is the function **FOO**. What is she doing?

```
(DE foo NIL
       (PRINT (APPEND (QUOTE (DE foo))
                      (CDR (GET (QUOTE foo)
                                (QUOTE EXPR)))))
       (PRINT (QUOTE (foo))))

(foo)
```

(GOOSSENS)

This is the function **BAR**. What is she doing?

```
(DE bar (l x)
       (IF (NULL l) x
           (bar (REVERSE (CDR l)) (CAR l))))
```

SEND MORE PUZZLES)

(REPORTS, MANUALS, BOOKS

AGRE P. , Functions as Data Objects : the Implementation of Functions in LISP, Computer Science Department, University of Maryland, TR-726, January 1979

CHAILLOUX J. , VLISP 8.2, Manuel de référence, Université de Paris-8-Vincennes, Décembre 1979

DYER M.G., FLOWERS M., MUCHNICK S. S., LISP/85 User's Manual, Computer Science Department, University of Texas, TR-77-4, November 1977

FLOWERS M., DYER M. G., MUCHNICK S. S., LISP/85 Implementation Report, Computer Science Department, University of Texas, TR-78-1, February 1978

GREUSSAY P., Le Système VLISP 16, Manuel de référence, Université de Paris-8-Vincennes et Ecole Polytechnique, Février 1979

HOFSTADTER D. R., GODEL, ESCHER, BACH : an Eternal Golden Braid, The Harvester Press, 1979

STEELE G. L., RABBIT : A Compiler for Scheme (A Study of Compiler Optimization), M.I.T. Artificial Intelligence Laboratory, AI-TR-474, May 1978

STEELE G. L., SUSSMAN G. J., The Art of the Interpreter, M.I.T. Artificial Intelligence Laboratory, AI Memo 453, May 1978

STEELE G. L., SUSSMAN G. J., Design of a LISP-based Processor, M.I.T. Artificial Intelligence Laboratory, AI Memo 514, March 1979

WEINREB D., MOON D. , LISP Machine Manual, M.I.T. Artificial Intelligence Laboratory, January 1979

WRITE MORE LISP REPORTS, MANUALS, BOOKS)

THE VLISP KIT : DESCRIPTON, IMPLEMENTATION and EVALUATION.

Jérôme CHAILLOUX

Département d'Informatique
Université de Paris 8 - Vincennes
Route de la Tourelle
75571 Paris Cédex 12

Décembre 1979

(in French)

ABSTRACT :

This study presents the realization of ***three systems*** VLISP (a dialect of LISP) developped at the University of Paris 8 - Vincennes, on the folowing machines :
- a 8 bit words micro-processor (Intel8080/Zilog80)
- a 16 bit words PDP-11
- a 36 bit words PDP-10

From these realizations is extracted an ***implementation model.***

Our study proposes a solution to the problems of construction and evaluation of such a system. These problems are :

1) The exhaustive description of the implementation. We propose a description based on the ***virtual, referential and prototype machine VCMC2*** .
2) The adequate representations of the VLISP objects and functions. We have associated some ***natural properties*** and we have established a ***functionnal typology.***
3) The efficiency of the interpreter (in words of core, execution time and ***power***). Our iterpreter does, for his own need, a optimal core allocation (in term of CONS module calls). The direct acces (which needs only one memory access) to the values of objects variable and function, and a type classification of functions allow a ***direct invocation*** of all typed functions.
4) The ***power*** of control structures. Our implementation's KIT generalizes the VLISP control structures SELF an ESCAPE, extends them with the new constructions ***EXIT, WHERE and LETF*** and unifies completly their description and implementation.

An incarnation of our model is given by the realization of a ***complete*** VLISP system in the referential machine VCMC2. The ***full*** code is given in appendix.

J. McCarthy

AN INTERESTING LISP FUNCTION

Ikuo Takeuchi (1978) of the Electrical Communication Laboratory of Nippon Telephone and Telegraph Co. (Japan's Bell Labs) devised a recursive function program for comparing the speeds of LISP systems. It can be made to run a long time without generating large numbers or using much stack. The program is

$$tak(x,y,z) \leftarrow \text{if } x \leq y \text{ then } y \text{ else } tak(tak(x-1,y,z), tak(y-1,z,x), tak(z-1,x,y)),$$

where the variables may range over the integers (including negative) or else over real numbers. The program has similar properties in the two cases, but proving termination seems more tedious if arbitrary real numbers are allowed. The program has several interesting features.

1. Inspection suggests that *tak* satisfies the equation

$$tak(x+a,y+a,z+a) = tak(x,y,z)+a,$$

whenever the computation terminates, and this can be shown by subgoal induction. Namely, it is true for the non-recursive case, and assuming it for the referred sets of arguments yields it for the main set. A formal proof can proceed along the lines suggested in (McCarthy 1978).

2. Experiment using LISP indicates first of all that the value of $tak(x,y,z)$ is always one of x, y or z. I don't see a proof of this fact in isolation.

3. Like the 91-function, the program computes a simple non-recursive function. Using LISP to compute some values of *tak* leads to the guess that it is the same as

$$tak0(x,y,z) = \text{if } x \leq y \text{ then } y \text{ else if } y \leq z \text{ then } z \text{ else } x.$$

Substitution shows that *tak*0 satisfies the functional equation for *tak*, and therefore by the *minimization schema* of (McCarthy 1978),

$$tak(x,y,z) = tak0(x,y,z)$$

whenever the former terminates. *A fortiori*, this establishes that $tak(x,y,z)$ takes one of the variables as value, but maybe that fact could be proved separately.

4. In order to prove that *tak* is total, we write a *"derived program"* $dtak(x,y,z)$ that computes the depth of recursion involved in computing $tak(x,y,z)$ using call-

by-value. We have

$$\begin{aligned} dtak(x,y,z) \leftarrow\ & \text{if } x \leq y \text{ then } 0 \\ & \text{else } 1 + \max(dtak(x-1,y,z), \\ & \qquad dtak(y-1,x,z), \\ & \qquad dtak(z-1,x,y), \\ & \qquad dtak(tak(x-1,y,z), tak(y-1,z,x), tak(z-1,x,y))). \end{aligned}$$

Experiment with LISP leads to the conjecture that for integer values of the variables, *dtak* is extensionally equivalent to

$$dtak0(x,y,z) = dtak00(x-y, z-y),$$

where

$$\begin{aligned} dtak00(m,n) = \ & \text{if } m \leq 0 \text{ then } 0 \\ & \text{else if } n \geq 2 \text{ then } m + n(n-1)/2 - 1 \\ & \text{else if } n \geq 0 \text{ then } m \\ & \text{else if } n = -1 \text{ then } (m+1)(m+2)/2 - 1 \\ & \text{else } (m-n)(m-n+1)/2 - m - 1. \end{aligned}$$

We don't bother to verify the conjecture. Instead we use *dtak*0 as a rank function to prove $\forall i.\Phi(i)$ by course-of-values induction where

$$\Phi(i) \equiv \forall xyz.(dtak0(x,y,z) = i \supset tak(x,y,z) = tak0(x,y,z)).$$

Since we already know that *tak*0 satisfies the functional equation of *tak*, we need only show that in the recursive case of *tak*, i.e. when $x > y$, the referred arguments are of lower rank than the main ones. Thus we must show that each of $dtak0(x-1,y,z)$, $dtak0(y-1,z,x)$, $dtak0(z-1,x,y)$, and $dtak0(tak0(x-1,y,z), tak0(y-1,z,x), tak0(z-1,x,y)$ is strictly less than $dtak0(x,y,z)$. Each inequality follows from a separate easy case analysis. Presumably termination could be proved for the non-integer case by a similar argument with a more complicated formula for *dtak*00. We leave this as an exercise for the reader.

5. Like Morris's program

$$morris(x,y) \leftarrow \text{ if } x = 0 \text{ then } 1 \text{ else } morris(x-1, morris(x,y)),$$

tak is more quickly computed by call-by-need (Vuillemin 1973). In fact the number of function calls appears to be exponential with call-by-value and quadratic with call-by-need. The culprit is the third argument of the outer call, namely $tak(z-1,x,y)$ which is often unneeded. Unlike *morris*, however, *tak* is total.

A call-by-need version of *tak* is given by

$$ntak(x, y, z) \leftarrow vtak\langle x, y, z\rangle,$$

where

$$\begin{aligned} vtak\, u \leftarrow\ & \textbf{if } numberp\, u \textbf{ then } u \\ & \textbf{else if n d } u \textbf{ then } vtak(\textbf{ a } u) - 1 \\ & \textbf{else } \{vtak \textbf{ a } u, vtak \textbf{ ad } u\}[\lambda xy. \\ & \quad \textbf{if } x \leq y \textbf{ then } y \\ & \quad \textbf{else } vtak\langle\langle x - 1, y, \textbf{ add } u\rangle, \langle y - 1, \textbf{ add } u, x\rangle, \langle \textbf{ add } u - 1, x, y\rangle\rangle]. \end{aligned}$$

vtak is obtained from *tak* in a somewhat ad hoc way. Since we don't always compute all arguments of a function we must work with triples whose elements are either numbers or applications of functions to arguments. The only functions that occur are *vtak* itself and subtracting one. Therefore, a number stands for itself, an application of *vtak* is represented by a triple, and an application of subtracting one is represented by a list of one element—the inner expression from which one is to be subtracted. Perhaps I'm being thick, but it seems to me that call-by-need requires lists or at least putting symbols on the stack.

MACLISP versions of *tak* and friends are in the file TAK.LSP[F78,JMC] at SU-AI. I thank Don Knuth for suggesting call-by-need. The external or publication LISP notation is as in (McCarthy and Talcott 1978). It has the following features: (1) **a** and **d** are used for *car* and *cdr*, and their compounds are formed similarly. (2) The infix . is used for *cons*, and $\langle x, y, \ldots, z\rangle$ is used for $list[x, y, \ldots, z]$. (3) $\{x, y, \ldots, z\}f$ is used for $f[x, y, \ldots, z]$ whenever we prefer to write the arguments first and the function name later.

References

(McCarthy 1978) John McCarthy, Representation of recursive programs in first order logic, *Proceedings of the International Conference on Mathematical Studies of Information Processing*, Kyoto, August 1978, 587–605.

(McCarthy and Talcott 1978) John McCarthy and Carolyn Talcott, **LISP: Programming and Proving**, preliminary edition, Computer Science Department, Stanford University, 1978.

(Takeuchi 1978) Ikuo Takeuchi, personal communication.

(Vuillemin 1973) Jean Étienne Vuillemin, *Proof techniques for recursive programs*, Ph.D. thesis, Computer Science Department, Stanford University, 1973.

A New Fast and Safe Marking Algorithm*

Toshiaki Kurokawa

Information Systems Lab., TOSHIBA R & D Center

Kawasaki, Japan

Abstract:

A new marking algorithm for garbage collection is presented. The method is a variation of the usual simple stacking algorithm. In practice, the new algorithm requires both less stack space and less time. One modification is to stack a node only when both the sublists are unmarked. The other innovation is a "stacked-node-checking" method invoked after each stack-overflow. With this method, a number of unnecessary nodes are eliminated, the stack is compacted, and the marking process can resume using the generated space in the stack.

Keywords and Phrases:

Marking algorithm, garbage collection, stack a node only when both sublists are unmarked, stacked node checking method, LISP

CR Categories: 4.19, 4.40, 4.49, 4.9

---

*) This work is in part supported by PIPS project of Electro-technical Laboratory, Ageney of Industry and Science, Ministy of International Trade and Industry, Japan.

Introduction:

Among existing marking algorithms, the simple stacking algorithm was known to be the fastest (see Knuth [1]). As shown in Alg. 1, it uses the stack space for storing the node elements whose cdr (right-link) is not yet attacked. Alg. 1 visits each node only once, which is the minimum number of visits. (A swapping algorithm, such as Schorr and Waite's [4] or Wegbreit's [5], visits each node twice.)

However, the number of stacking operations (push and pop) cannot be said to be minimum in Alg. 1.

The algorithm proposed here is a variation of this simple stacking algorithm and is faster, because a node is stacked only when both the sublists are unmarked. Thus, a number of stacking operations are eliminated.

The other problem on the marking algorithm is the amount of necessary working space. In this point, Wegbreit's algorithm is the best known. Generally speaking, stacking algorithms are worse than swapping algorithms, because a staking algorithm needs up to the total number of nodes in the worst case. If the stack space is less, stack overflow occurs.

Some escape methods have been proposed to handle this stack overflow. For the algorithm proposed here, the stacked-node-checking (SNC) method is employed, which deletes the unnecessary nodes from the stack. Some of the stacked nodes are unnecessary because the sublists are already marked, or one of its sublists is already marked

and the other one can be traced and marked. After SNC, the stack space is compacted by the elimination of these unnecessary stacked nodes, and the marking process is resumed using the generated space.

This SNC process is so simple that the total execution time, including many stack overflows, is faster than that of Schorr-Waite's swapping algorithm.

This fast marking algorithm was developed and implemented for LISP1.9 garbage collector [3].

Fast execution:

The problem for the tree marking (or the tree tracing) is, in short, a problem concerning the determination points (or branch points). The simple stacking algorithm (Alg. 1) uses the stack space to store the branch node. The swapping algorithm, on the other hand, uses the extra mark bit to indicate that this node is a branch point and its cdr link is not yet traced. In the swapping algorithm, the pointers are dynamically changed to keep track of the process history. To recover this destruction, each node must be visited twice. On the other hand, the simple stacking algorithm visits each node only once, which is the minimum number of visits. This difference in the amount of visiting makes the simple stacking algorithm faster than the swapping algorithms.

However, the number of stacking operations (push and pop), which occur once for each node in the simple stacking, is not the minimum. For example, examining Car Tree (Fig. 1), the marking can be easily accomplished without stacking, because there is eventually no decision

point in this tree.

The proposed algorithm (Alg. 2) uses two local variables, car-node and cdr-node, to check whether or not the node is really a branch node. The car-node and the cdr-node are both list elements and both unmarked. The push-stack operation occurs only when such a branch node is attacked.

On the contrary, the simple stacking algorithm (Alg. 1) pushes every encountered node.

This technique was hinted at from Wegbreit's bit-stacking algorithm [5]. There are two improvements. One involves checking the node to determine if it is a list-element or an atom-element. This algorithm checks whether it is an unmarked list element or not. The other improvement is that this algorithm uses stacking, while Wagbreit used pointer swapping. These improvements have made this fast algorithm both faster and require less space.

Table 1 shows the execution results for special cases. Table 2 shows the results for typical LISP programs. Remarkably that, the used stack space has reduced to less than half of that required by simple stacking.

Some extensions:

The introduction of two local variables (car-node and cdr-node) has enabled this stack space reduction. Is it possible to make more reduction if other local variables are added, such as caar-node, cdar-node, etc. ?

It is easily shown that stack depth reduction can be realized through introduction of other local variables. For example, using caar-node and cdar-node, the push stack operation can be postponed until such a time as all of the three nodes — caar-node, cdar-node and cdr-node — are known to be unmarked list elements. Algorithm 3 represents this modification for case 4 in Alg. 2. Stack reduction gain is 1, because the effect is realized just before the bottom of the tree.

Considering that only 10 or 11 stack spaces are used in typical LISP programs, another 22 local variables could delete the necessity for the stack space.

As compared to this explicit gain in stack reduction, the speed up gain will not be so clear. The reason is that the percentage of the stack operation time in the total marking process time has already become small, through the first introduction of car-node and cdr-node. Another reason is that the variable assignments cost some time proportional to the number of local vairables, and it will lessen the gain realized from deleting stack operation time.

SNC method:

The weakpoint of a stacking algorithm is the theoretical need for large stack space. By the worst case analysis, a space as large as the total number of nodes is needed. The swapping algorithm, however, requires only the extra bit space which will cost (the-node-space/N) where there are N bits in a word (usually, there are 32 or 36).

In actual marking algorithm applications, this stacking algorithm weakness will not be realized because, usually, the situation is much better. Only about 30 words for stack space is sufficient.

In the above better case, the stacking algorithm can be said to be better than the swapping algorithm, even from the work storage viewpoint.

To solve the worst case problem, several methods were proposed to handle the stack overflow.

Schorr and Waite [4] proposed to use the swapping algorithm after the stack overflow. Kurokawa [2] proposed to move the tracing information from the stack to the tree itself, using extra mark bits. In this fast marking algorithm, Kurokawa's method cannot be applied because there is no node relation in the stack. Schorr and Waite's proposal is highly appropriate, but it is too expensive, because it means that two marking algorithms — stacking and swapping — co-exist at the same time and that both stack space and extra bit space are necessary.

The SNC method, proposed here, was invented for this fast marking algorithm. It uses no extra bits. First, each stacked node is checked to determine if it is necessary or unnecessary. Unnecessary nodes are then deleted from the stack. The space is made to resume the marking process.

Algorithm 4 shows the SNC method after the stack overflow. The first loop means stacked node checking. The function checks(shown

in Alg. 5) returns 0 when both the sublits are marked and unnecessary. It returns the node element (which may be different from the stacked element) when both of its sublists are unmarked. It should be noted that checks function is another kind of marking process and that the content of the stack will be changed.

The next loop deletes unnecessary cells and tries to make space for the marking process. If there is no space to be had, it gives up and advises of a fatal stack overflow error. If there is space to work in, the marking process resumes.

After the marking process successfully ends, the last loop begins to utilize the "checked" nodes, which are stacked at the first checking loop. The whole marking process ends itself after all the "checked nodes" are cleared from the stack.

The main SNC method characteristic is its speed. Even after a number of overflows, it is faster than the swapping algorithm, as shown in Table 3. If the overflow occurs only once or twice, the speed is faster than that of the simple stacking algorithm.

The SNC method defect is that it cannot ensure that no stack space is necessary. In a typical LISP program, where an 11 word stack is necessary without stack overflow, the SNC method can be successfully applied only after a 10 word stack is ensured. However, if stack space are used more, such as in the typical examples in Table 3, the SNC effect becomes very large.

Concluding remarks:

The fast marking algorithm was implemented for LISP1.9 [3] in the autumn of 1975. In LISP1.9, the marking process occupied about 3/4 of garbage collection time. The effects of this fast marking, both on speed and on stack space, are quite large. 20% speed-up gain is realized and only half of the stack space is necessary for the usual LISP programs. The old marking was by a simple stacking algorithm.

The SNC method, on the other hand, is not employed for typical LISP programs. In a sense, this is very pleasant for the fast marking algorithm, because it can be said to be better than others, even from the working space standpoint. The SNC method is a kind of "emergency exit" method for the fast marking algorithm (it can be applied to the simple stacking case, though). It would be better if the emergency exit were not used any more.

Reference

1. Knuth D. E., Fundamental Algorithms, The Art of Computer Programming 1, Addison-Wesley (1968)
2. Kurokawa T., New Marking Algorithms for Garbage Collection, Proc. of 2nd. USA-JAPAN Computer Conference (Aug. 1975).
3. Kurokawa T., LISP1.9 Programming System, Journal of Information Processing Society in Japan ~~(in Press)~~ vol 17, no. 11, 1056–1063 (1976)
4. Schorr H. and Waite W. M., An efficient machine-independent procedure for garbage collection in various list structures, C.ACM. 10, pp. 501-506, (Aug. 1967)

5. Wegbreit B., A space-efficient list structure tracing algorithm, IEEE Trans. Comp. C-21, 9, pp. 1009-1010, (Sep. 1972)

6. Goldman N. M., Sentence Paraphrasing from a Conceptual Base, C. ACM, 18, 96-106, (Feb. 1975)

```
Procedure  Simple_stack_mark (node);
  begin
    if   marked (node) then return ( )
                       else push (bottom-mark);
    loop {repeats until return ( ) is executed }
         mark (node);
         push (node);  { Stack-overflow will occur here. }
         node: = car[node] ;
         if  marked (node) then
                loop  { repeats poping the stack
                                until bottom or unmarked-
                                element is encountered }
                   node: = pop ( );
                   if   node = bottom-mark
                              then return ( ) endif ;
                   node: = cdr[node] ;
                   if   marked (node)
                              then  exit-loop ( )  endif ;
                endloop
         endif
    endloop
  end
```

Alg. 1 Simple_stack_marking algorithm

In this and following algorithms, marked (node) means that either the node is already marked according to the list element or it is an atom (i.e. non-list element).

```
Procedure    Fastmark (node);
  beign   localvar: car-node, cdr-node;
    if    marked (node) then return ( )
                  else push (bottom-mark);
                       mark (node)   endif ;
    loop  { repeats until return ( ) is executed }
          car-node: = car[node] ;
          cdr-node: = cdr[node] ;
        { The following 4 cases exist for car-node and cdr-node. }
    case 1: { Both car and cdr are already marked.
              Pop stack will occur only for this case. }
      if marked (car-node) ∧ marked (cdr-node)
        then   node: = pop ( ) ;
          if  node  = bottom-mark then return( ) endif
      endif;
    case 2: { Only car-node is already marked.
              Marking will continue without push or pop. }
      if marked (car-node)∧ ¬ marked (cdr-node)
        then  mark (cdr-node);
              node: = cdr-node
      endif ;
```

(Alg. 2 Fast marking algorithm) cont.

```
case 3:  { Only cdr-node is already marked.
           Marking will continue without push or pop. }
   if   ¬ marked (car-node) ∧ marked (cdr-node)
      then mark (car-node);
           node: = car-node
   endif ;
case 4:  { Both car-node and cdr-node are unmarked.
           Push stack occurs and the stack-overflow may be
           caused here. }
   if   ¬ marked (car-node) ∧ ¬ marked (cdr-node)
      then  mark (car-node);
            mark (cdr-node);
            push (cdr-node);
            node: = car-node
   elseif                    endif
endloop
end
```

Alg. 2 Fastmark algorithm

(Case 4 is modified in Alg. 3)

```
        case 43:  { Only cdar-node is marked.  Loop again. }
        if ¬ marked (caar-node)  ∧ marked (cdar-node)
              then    mark (caar-node);
                      car-node: = caar-node
        endif ;
        case 44:  { Both are un-marked.  Push-stack occurs. }
        if  ¬ marked (caar-node)∧ ¬ marked (cdar-node)
              then    mark (caar-node);
                      mark (cdar-node);
                      push (cdr-node);
                      cdr-node: = cdar-node;
                      car-node: = caar-node
        endif
    endloop
  end
endif
```

Alg. 3 Modified Fastmark algorithm on the part of case 4.

```
case 4:  { Expanded using caar-node and cdar-node.
           Stack depth is decreased by 1. }
  if  ¬ marked (car-node)∧ ¬ marked (cdr-node) then
    begin  local vor : caar-node, cdar-node;
           mark (car-node);
           mark (cdr-node);
           loop {repeat until exit-loop( ) is executed }
              caar-node: = car[car-node] ;
              cdar-node: = cdr[car-node] ;
           case 41:  { Both are marked.  No push is necessary. }
           if    marked (caar-node) ∧ marked (cdar-node)
                 then  node: = cdr-node;
                       exit-loop ( )
           endif ;
           case 42:  { Only caar-node is marked.  Loop again ! }
           if    marked (caar-node)∧ ¬ marked (cdar-node)
                 then  mark (cdar-node);
                       car-node: = cdar-node
           endif ;
```

(Alg. 3 Modified Fastmark algorithm on the part of case 4.) cont.

```
Procedure  Stacked-node-check (car-node, cdr-node);
    { This procedure is invoked when a stack-overflow occurs.
      Thus stack-ptr = top-of-stack-space; you cannot push anymore. }
  begin  localvor: write-ptr, answer:
    loop  { repeats checking nodes until bottom-of-stack is encountered. }
        node: = pop ( );
        if  node = bottom-mark then exit-loop ( ) endif ;
        answer: = checks (node);
        { answer is either 0 (already marked) or tree-node }
        write-to-stack (answer);
        { using stack as 1-dimensional array and replace the
          stacked node by 0 or node-element. }
        { The node-element means that both its car and cdr were
          un-marked. }
    endloop;
      { The stacked elements may be changed. Some of them may
        be 0, which means that it can be deleted.
        The stack-ptr is now at the bottom. }
        write-ptr: = stack-ptr;
    loop  { deletes the 0 nodes until top-of-stack-space is encountered }
        stack-ptr: = stack-ptr + 1 ;
        if  stack-ptr = top-of-stack-space
              then exit-loop( ) endif ;
        node: = read-stack ( );{read-stack returns the stacked-node
                                which is at the stack-prr position.}
```

```
    if node = 0 then   write-ptr: = write-ptr + 1 ;
                       write-to-stack (node) { write-to-stack
    endif ;                  writes the node into the stack. }
endloop;
{ After this deletion, stack-ptr = top-of-stack, and write-ptr =
   top-of-necessary-nodes. }
if  write-ptr = top-of-stack
        then error-of-fatal-stack-overflow ( ) endif ;
{ Otherwise, there are some spaces for pushing cdr-node. }
stack-ptr: = write-ptr;
push (cdr-node);  { Once overflow occured due to this operation. }
Fastmark (car-node);  { Continue marking. }
node: = pop ( );  { This node is actually old cdr-node. }
Fastmark (node);
loop  { recovers the stacked nodes,until they exhaust. }
    node: = pop ( );
    if  node = bottom-mark then return ( ) endif ;
    { Otherwise, both car and cdr must be marked.
      However, car[node] and cdr[node] are already
      marked by the function checks [node]. }
      car-node: = car[node] ;
      cdr-node: = cdr[node] ;
      push (cdr-node);
```

```
        Fastmark (cdr-node);
        node: = pop ( );
        Fastmark (node)
    endloop
end
```

Alg. 4. Stacked-Node-Checking method.

```
Function    checks (node);
  begin   localvar: car-node, cdr-node;
    loop  {repeats until either return (∅) or return (node) is executed.}
        car-node: = car[node];
        cdr-node: = car[node];
        if   marked (car-node) ∧ marked (cdr-node)
             then return (∅) endif ;   { Unnecessary node ! }
        if   marked (car-node) ∧ ¬ marked (cdr-node)
             then    mark (cdr-node);
                     node: = cdr-node
        endif ;
        if ¬marked (car-node)  ∧ marked (cdr-node)
             then    mark (car-node)
                     node: = car-node
        endif ;
        if   marked (car-node)    ∧ marked (cdr-node)
             then    mark (car-node);
                     mark (cdr-node);
                     return (node)   { Cannot chase from this node.
                                       This node must exist in the stack. }
        endif ;
    endloop
  end
```

Alg. 5 Function — check node

B-Tree	N = 8192		N = 16384	
	Time (msec)	Stack-Used (words)	Time (msec)	Stack-Used (words)
This algorithm	252	14	510	15
Simple-stacking	259	15	523	16
Pointer-swapping	714	0	1414	0

Car-Tree	N = 8192		N = 16384	
	Time (msec)	Stack-Used (words)	Time (msec)	Stack-Used (words)
This algorithm	289	0	574	0
Simple-stacking	349	8192	671	16384
Pointer-swapping	746	0	1547	0

Revised-Car-Tree	N = 8192		N = 16384	
	Time (msec)	Stack-Used (words)	Time (msec)	Stack-Used (words)
This algorithm	356	1	709	1
Simple-stacking	388	8192	773	16384
Pointer-swapping	835	0	1677	0

Pseudo-Car-Tree	N = 8192		N = 16384	
	Time (msec)	Stack-Used	Time (msec)	Stack-Used
This algorithm	362	2730	731	5461
Simple-stacking	398	8192	791	16384
Pointer-swapping	840	0	1670	0

Table 1. Comparison of several data on marking algorithms.

N means the total free list elements number.

	N = 30780	
	Time (msec)	Stack-Used (words)
This algorithm	803	11
Simple-stacking	974	26
Pointer-swapping	1896	0
Revised fast marking (caar-node and cdar-node Alg. 3)	799	10
Limited stack SNC-method used (overflow count = 6)	862	10

Table 2. Results for typical LISP program.

The program is BABEL, written by Goldman [6], which is one of the largest programs on LISP1.9.

N means the total free list elements number.

Pseudo-Car-Tree (N = 16384)	Stack-Area Allocated	Time (msec)	No. of Overflow
This algorithm	∞ (5461)	731	0
	4000	769	1
	1000	790	5
	250	809	22
	50	849	113
Simple stacking	∞ (16384)	791	0
Pointer-swapping	0	1670	0

Ladder (N = 16384)	Stack-Area Allocated	Time (msec)	No. of Overflow
This algorithm	∞ (8192)	629	0
	2000	697	4
	500	707	16
	125	722	66
	25	878	356
	10	1201	1023
	3	5146	8190
Simple stacking	∞ (8192)	651	0
Pointer-swapping	0	1542	0

Table 3. Marking time when stack-space is limited.

N means the total free list elements number.

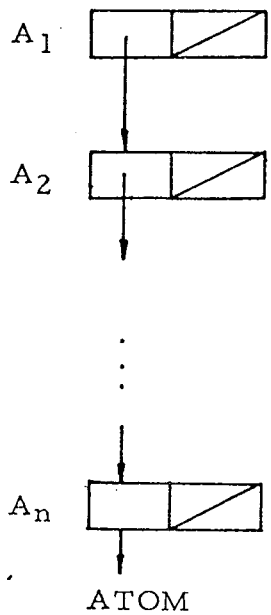


Fig. 1 Car-Tree

(Worst case for simple

stacking algorithm.)

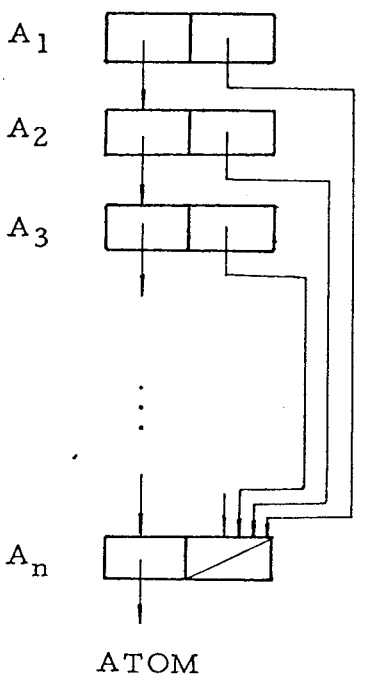


Fig. 2 Revised-Car-Tree
(Worst case for Wegbreit's
algorithm.)

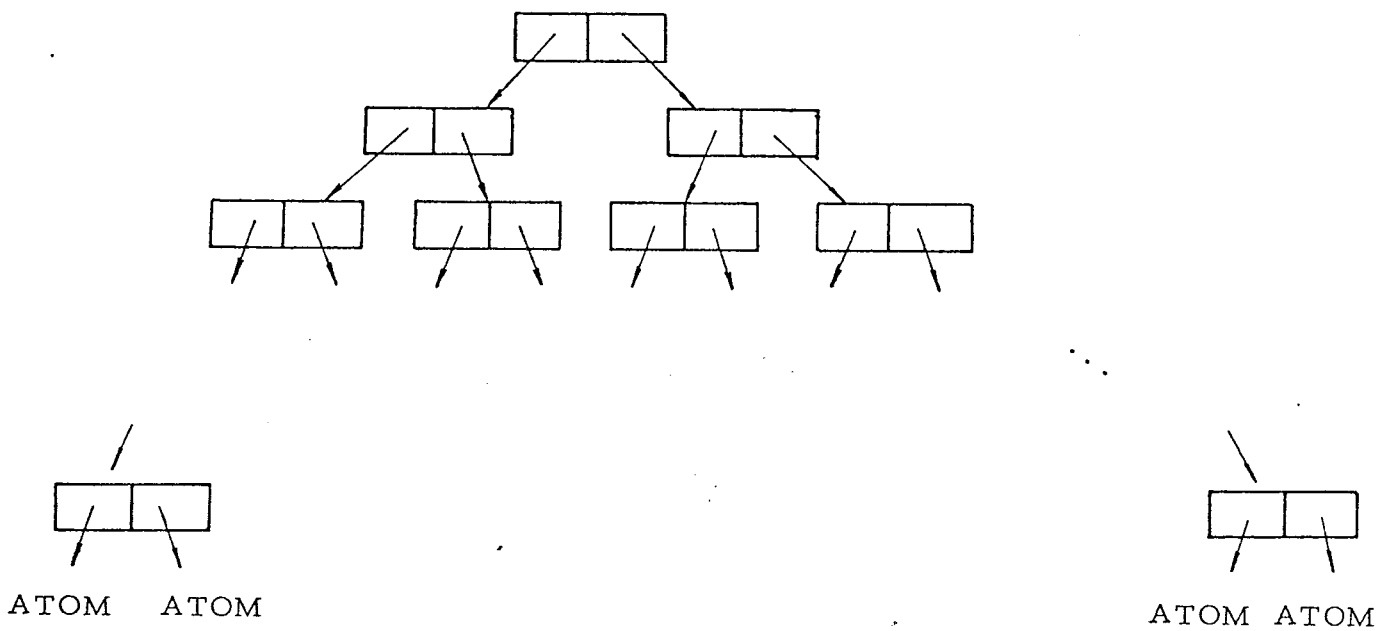


Fig. 3 B-Tree

(Difference is least between this algorithm and the simple stacking algorithm for this case.)

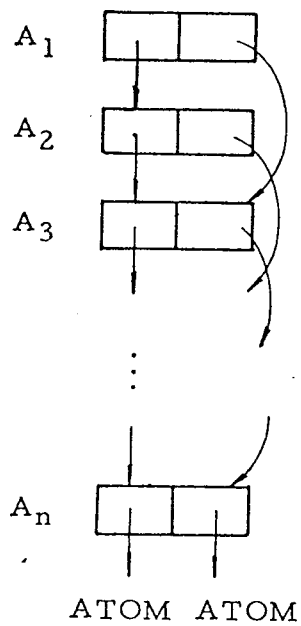


Fig. 4 Pseudo-Car-Tree

(This algorithm can work with only one third or the stacking space, compared with the simple stacking algorithm.)

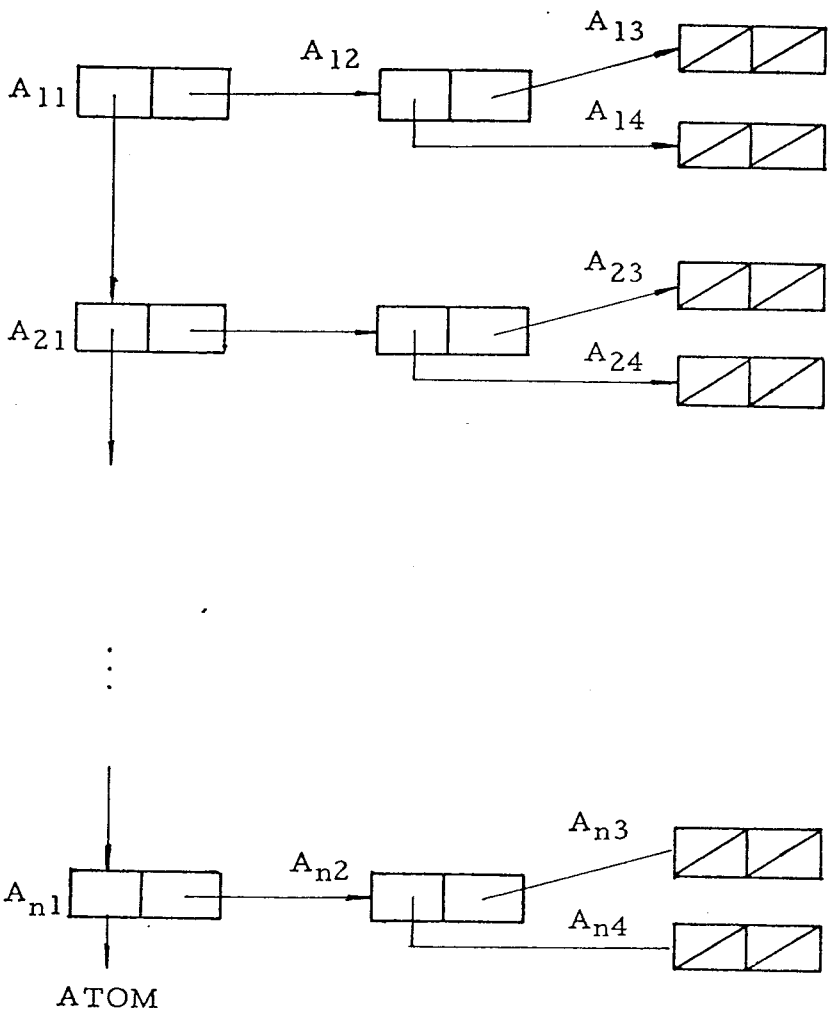


Fig. 5 Fork Case

(Worst case for this marking algorithm
If the stack space is less than n, the
marking process cannot be accomplished.)

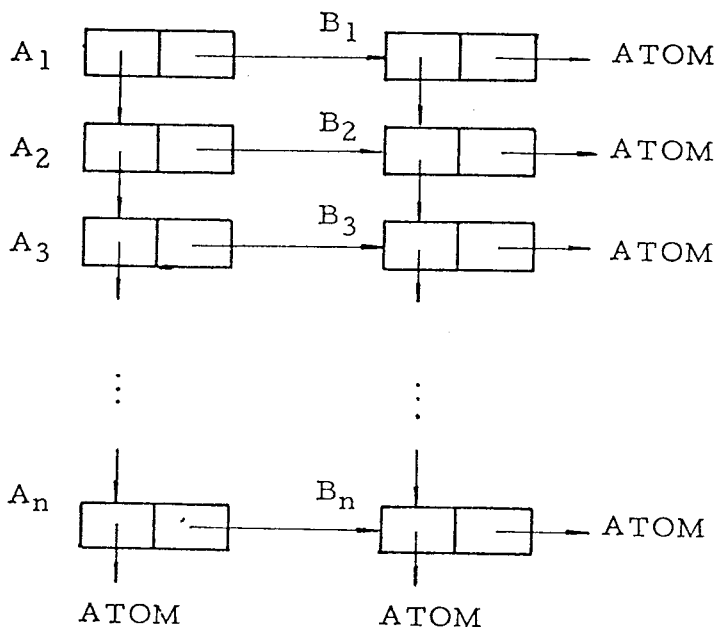


Fig. 6 Ladder Case

(This data needs n-words stack space for this marking algorithm. Using the SNC-method, however, the algorithm can work with only a 3-word stack space.)

MECHANICAL CONSTRUCTION OF A NEW EFFICIENT FLATTEN.

Yves Kodratoff,+ Jean-Pierre Jouannaud++

+ G.R. 22 du C.N.R.S., Institut de Programmation 4 Place Jussieu 75230 PARIS CEDEX 05.

++ Université de Nancy 2, 42 Avenue de la Libération 54000 NANCY.

1. MCFLAT.

An efficient function FLATTEN is "well-known" and due to Mc CARTHY[5]. We give below a version of it, using the predicates NULL and ATOM.

```
(DE MCFLAT(X)(FOO X NIL))
(DE FOO(X Z)(COND
              ((NULL X)Z)
              ((ATOM X)(CONS X Z))
              (T(FOO(CAR X)(FOO(CDR X)Z)))))
```

In order to study the complexity of this function we shall count the number of times FOO is called when one evaluates (MCFLAT X) where X is a list containing n atoms and m couples of parenthesis. In an intuitive way, consider first the list (A) which leads to 3 calls of FOO (the initial one and then 2 recursive calls). When an atom is added to it , say X becomes (A B), 5 calls to FOO are necessary, i.e. we add 2 new calls to FOO per new atom. The same is true when one considers ((A)), i.e. we add 2 calls by new nesting level. This means that FOO is called 2n+2m-1 times by MCFLAT.

2. MQFLAT : transforming programs contructed from examples.

2.1.-

Our methodology for getting programs from input-output examples is described in[1, 2, 3] . In principle, it consists into 4 steps.

The input sequence is transformed into a predicate sequence, the output sequence is tranformed into a computation traces sequence, each of the new sequences is transformed to sets of recursion relations which can finally be proved[2, 6]to be equivalent to a program.

We shall study here the example of the top-level COPY function. The input-output sequence is $\{x_o=(A)\rightarrow f(x_o)=(A);\ x_1=(A\ B)\rightarrow f(x_1)=(A\ B);x_2=(A\ B\ C)\rightarrow f(x_2)=(A\ B\ C);...\}$.

We deduce from it the following predicates and computation traces sequences:

$\{p_o(x)=(ATOM(CDR\ X))\rightarrow f_o(X)=(CONS(CAR\ X)NIL);$

$p_1(x)=(ATOM(CDDR\ X))\rightarrow f_1(X)=(CONS(CAR\ X)(CONS(CADR\ X)NIL));$

$p_2(X)=(ATOM(CDDDR\ X))\rightarrow f_2(X)=(CONS(CAR\ X)(CONS(CADR\ X)(CONS(CADDR\ X)NIL)));\ ...\}$.

Matching p_i and p_{i+1} leads trivially to the recursion relation $p_{i+1}(X)=p_i(CDR(X))$.

Matching f_i and the "tail" of f_{i+1} (as the arrows show) leads to the recursion relation: $f_{i+1}(X)=CONS(CAR(X), f_i(CDR(X)))$ which we shall rather write for further transformation purposes : $f_{i+1}(X) = h(X, f_i(CDR(X)))$ where $h(X, Z) = (CONS(CAR\ X)Z)$.

We finally obtain a program whose stopping conditions are given by the first predicate and the first trace and whose recursive body is given by the recursion relations:

```
(DE COPY(X)(COND
          ((ATOM(CDR X))(CONS(CAR X)NIL))
          (T(H X(COPY(CDR X))))))
(DE H(X Z)(CONS(CAR X)Z))
```

2.2.

The programs we thus obtain are generally defined for a specific input domain which is here the flat lists(they have only atoms at their top-level). As a matter of fact, it can be proved that they are defined for any list but they act at the top-level only. We have described in [4a] how to obtain from this program F an other program F' which is recusively defined on all nesting levels of the list. We describe in [4b] how to obtain directly (FLATTEN(F' X)), i.e. a program which executes the action of F at all nesting levels and flattens its result.

We shall not detail here these transformations : they are quite cumbersome to describe in the general case. From the intuitive point of view, we can simply say that the form of the synthesized programs allowsthe reasoning : F' is like F except that it tests if (CAR X) (or other $(CAD^kR\ X)$ expressions) in the stopping branch are atomic or not. If yes then F' is identical to F, if not then F' calls itself again, applied to the corresponding non-atomic expression.

Applied to COPY, this transformation leads to MQFLAT:

```
(DE MQFLAT(X)(FQQ X NIL))
(DE FQQ(X Z)(COND                                          FQQ1
          ((ATOM(CDR X))(COND
                    ((ATOM(CAR X))(CONS(CAR X)Z))          FQQ2
                    (T(FQQ(CAR X)Z))))
          (T(HQQ X(FQQ(CDR X)Z)))))                        FQQ3
(DE HQQ(X Z)(COND
          ((ATOM(CAR X))(CONS(CAR X)Z))                    FQQ4
          (T(FQQ(CAR X)Z))))
```

When (MQFLAT '(A)) is evaluated, only 1 call to FQQ1 is needed. When one adds one atom in the list then (MQFLAT '(A B)) calls FQQ1 then HQQ (thus FQQ4), i.e. each new atom induces 2 new recursive calls. When one adds a nesting level, MQFLAT applied to ((A)) needs a call to FQQ1 and a call to FQQ2 ,i.e. each nesting level adds only 1 supplementary recursive call. It follows that if x contains n atoms and m pairs of parenthesis, 2n+m-2 calls to FQQ and HQQ are needed.

2.3. An other(inefficient) FLATTEN.

From the above example , one might believe that we implicitely claim that our methodology leads always to good programs. This would be wrong and we feel it interesting enough that automatically constructed programs are not automatically the worse possible!

One can get an other COPY from the examples by matching "root to root" f_i and f_{i+1}.

The recurrence relations are slightly more complicated since one needs to generalize $f_i(X)$ to $g_i(X, NIL)$ and look for recursion relations between $g_i(X,Z)$ and $g_{i+1}(X,Z)$ where $g_i(X,Z)$ is the same expression as $f_i(X)$ except that the NIL of $f_i(X)$ is replaced by Z.

Once this generalization is made, one finds quite easily the recursion relations :

$f_i(X)=g_i(X,NIL)$;

$g_o(X,Z)=(CONS(CAR\ X)Z)$, $g_{i+1}(X,Z)=g_i(X,h_i(X,Z)$;

$h_o(X,Z)=(CONS(CADR\ X)Z)$, $h_{i+1}(X,Z)=h_i(CDR(X),Z)$.

One also remarks that the X of g_i recurs as $X \to X$ while the X of p_i (the domain predicates) recurs as $X \to (CDR\ X)$ so that we need another variable (we shall call it XX) in order to express the domain recurrence relations.

Finally, we obtain the program :

```
(DE COPY2(X)(G2 X X NIL))
(DE G2(X XX Z)(COND
                ((ATOM(CDR XX))(CONS(CAR X)Z))
                (T(G2 X(CDR XX)(H2 X XX Z)))))
(DE H2(X XX Z)(COND
                ((ATOM(CDDR XX))(CONS(CADR X)Z))
                (T(H2(CDR X)(CDR XX)Z))))
```

We apply the same transformation to this new COPY and find :

```
(DE MKFLAT(X)(FKK X X NIL))
(DE FKK(X XX Z)(COND
                ((ATOM(CDR XX))(COND
                                ((ATOM(CAR X))(CONS(CAR X)Z))
                                (T(FKK(CAR X)(CAR X)Z))))
                (T(FKK X(CDR XX)(HKK X XX Z)))))
(DE HKK(X XX Z)(COND
                ((ATOM(CDDR XX))(COND
                                ((ATOM(CADR X))(CONS(CADR X)Z))
                                (T(FKK(CADR X)(CADR X)Z))))
                (T(HKK(CDR X)(CDR XX)Z))))
```

MKFLAT is clearly of degree 2 polynomial complexity, it runs quite slowly as compared with the other two.

One should notice that the good performance of MQFLAT originates from the fact that the H function which embedds COPY is a constant function while the H2 function embedded in G2 is not constant. This is a very special case and one should in the general case choose the "root to root" matching rather than the matching of section 2.1..

3. Discussion.

It is difficult to say that our program is "better" than Mc CARTHY's since the stopping conditions are not the same and since we use cross-recursion. We therefore do not claim that each LISP system should implement MQFLAT. It is nevertheless surprising that a transformation which is systematic enough to be implemented(and shall be soon implemented) , i.e. a program that can be automatically synthesized can well challenge the programs due to very skillful programmers.

This illustrates a point which a matter of deep contests among the practisers of Artificial Intelligence : does A.I. rely or not on simulation of behaviour ?

It is clear that we never analysed our own way of programming in order to discover MQFLAT . On the contrary , this program comes from considerations on necessities internal to program synthesis by machine. In the AISB society we must admit that we have a tendency to accentuate the AI rather than the SB.

Aknowledgment : We were induced to this work through discussions with J S. MOORE.

REFERENCES.

[1] J.-P. JOUANNAUD, Y. KODRATOFF " Characterization of a class of functions synthesized from examples by a SUMMERS like method using a "BMW" matching technique" Proc. IJCAI-79 Tokyo(1979) and Pub. Univ. Nancy C.R.I.N. (1979).

[2] Y. KODRATOFF " A class of functions synthesized from a finite number of examples and a LISP program scheme", Interntl. J. of Comp. and Information Sciences Vol 8, n°6, (1 9 7 9).

[3] Y. KODRATOFF, J. FARGUES "A sane algorithm for the synthesis of LISP functions from examples : the BOYER and MOORE algorithm", Proc. AISB meeting , Hamburg (1978), pp 169-175.

[4] a - Y. KODRATOFF, J.-P. JOUANNAUD " Construction automatique à partir d'exemples de fonctions de listes récursivement définies pour tous les niveaux d'imbrication des listes" Actes congrès AFCET/IRIA Reconnaissance des formes et Intelligence Artificielle Toulouse (1979).

b - Y. KODRATOFF, J.-P. JOUANNAUD "The transformation of top-level defined to all nesting levels defined list functions" Pub. Univ. Nancy, C.R.I.N. (1979).

[5] cited by J S. MOORE " A tour through a working theorem prover", 4th Workshop on Artificial Intelligence Bad Honnef (1979).

[6]P.D. SUMMERS " A methodology for LISP program construction from examples" J. ACM (1977), 24, 161-175.

Does LISP differ from ALGOL essentially ?

Friedemann Simon

Institut für Informatik
Universität Kiel
Olshausenstr. 4o - 6o
D-23 Kiel

A fare spread opinion is that LISP and ALGOL belong to different "families" of programming languages. In our current activities concerning LISP, we are trying to characterise pure LISP as an ALGOL-like programming language in the sense of ALGOL 6o resp. ALGOL 68 /1/, /2/. LISP is considered as a sublanguage of ALGOL 6o, where the datatype "s-expression" with its 5 standard functions is introduced and where procedure identifiers are allowed as values of function-procedures (ALGOL 68) in order to have upward FUNARGs /3/. In contrast to the operational semantic definitions via interpreters, this approach gives a precise, mathematical definition of the LISP-semantic. Within this framework we are able to prove properties of LISP-programs much easier than using inductive proofs based on an interpreter. Our method for modelling variable bindings follows the well known ALGOL 6o definitions, which are very close to the FUNARG-feature of LISP 1.5, while other authors prefer the "shallow access binding" method; e. g. M.J. Gordon gives a formal definition of pure LISP by algebraic methods /4/.

LISP- and ALGOL 6o implementations have an important difference concerning their run-time storage management. ALGOL 6o only needs a stack (deletion strategie), while LISP requires a heap (retention strategie) in order to keep variable bindings and data i. e. s-expressions. Starting from a proof scetch given by M.J. Fischer /5/ it has been shown that every LISP-program π can be transformed in a functional equivalent program π', which is deletion-tolerant, and where the operator cons is only used to construct a non-atomic final result of π' from its atoms /1/.

A simple consequence of the proof is that cons-free LISP with FUNARGs (c. f. LISP 1.5) is universal. If we restrict ourselves to cons-free LISP without FUNARGs, it can be shown using the theory of stack automata that this sublanguage is not universal /6/. The universality of LISP is given either by cons (neccessary to implement an interpreter) or in cons-free LISP by downward FUNARGs (procedure calls as in ALGOL 6o); upwards FUNARGs are irrelevant for the universality.

/1/ Simon,F.:Zur Charakterisierung von LISP als ALGOL-ähnliche Programmiersprache mit einem strikt nach dem Kellerprinzip arbeitenden Laufzeitsystem. Bericht 2/78, Institut für Informatik und praktische Mathematik der Universität Kiel, 1978

/2/ Lippe,W.M.,Simon,F.:A mathematical semantic definition for LISP. to appear

/3/ Sandewall,E.:A Proposed Solution to the FUNARG Problem. Uppsala Univ.-Dept. of Comp. Sc., Rep. 29, 197o

/4/ Gordon,M.J.C.:Models of Pure LISP (a worked example in semantics). Dept. of Machine Intelligence, Univ. of Edinburgh, Rep.31, 1973

/5/ Fischer,M.J.:Lambda Calculus Schemata. SIGPLAN Notices 7(1),1972

/6/ Simon,F.,Trademann,P.:Eine Beziehung zwischen cons-freiem LISP und Stackautomaten.Elektronische Informationsverarbeitung und Kybernetik EIK 14, Nr.12, 1978

LISP HISTORY

Herbert Stoyan
University of Dresden
Togliattistr. 40
806 Dresden
DDR

For the SIGPLAN conference on history of programming languages held in Los Angeles in this June, J. McCarthy had to write a paper about LISP-history (1). He was very able to do this because he has given a talk on LISP history in summer 1974 at M.I.T. (2) and has contributed since then a lot of remarks and comments to my work on compiling a complete history of our language. His paper corresponds to the state of our knowledge in May of this year (1978) before D. Park found the original LISP 1 manual (3).

Using this material I can now give a better description of the old matters. But there are open questions too - the informed reader is kindly asked to help in answering them.

a) PREHISTORY

J. McCarthy was born in 1927 and studied Mathematics. In 1948 he became BS at the CalTech and in 1951 he graduated with a PhD on differential equations from Princeton University. His interest in problems of A.I. came from an accidental presence at a symposium on cybernetics held at CalTech in 1949. Here he heard J. Von Neumann and other pioneers in this field. Since then he worked among other things in this field and wrote a paper on the inverting of Turing-machines to summarize some of his thought. The paper was published later 1956 in the collection "Automata Studies" (4).

In 1952 C. Shannon invited J.Mc Carthy and M. Minsky to work in the Bell Telephone Labs where they worked together during that summer. Topics of discussion were questions of cybernetics and automata theory. In the time until 1956 McCarthy, Shannon, Minsky and some other people came to form the opinion that A.I. could be reached by using digital computers rather than by cybernetic vehicles.

When McCarthy became Assistant Professor at Dartmouth College, he organized a summer school on cybernetics and artificial intelligence (5). He asked the Rockfeller Foundation for money, writing a proposal that would, as he found in 1977, make sense today, too.

A lot of researchers came to visit the conference, ten were more or less present the entire time. Very important was the discussion with A. Newell and H. Simon. Both had some experience with the use of

electronic computers at this time (at the JOHNNIAC at RAND corporation) and were developping their "logic theory machine" (6). For this work they proposed a language for formulating the logical means in their system - the "logical language" LL. Key idea of this language was the concept of lists.

IPL (information processing language), as they called the language later, was more an assembly language for artificial intelligence and J. McCarthy didn't like it very much. He hoped to have, in a language of this kind, algebraic expressions as he probably saw in FORTRAN, which was, 2 years after its announcement (7), running in the spring of 1956 (8).

b) EARLY HISTORY

But it is clear and well known which influence IPL had on the further development of AI and of programming languages (9, 10). Now it was clear for McCarthy : his language must be of FORTRAN style but doing list-processing as IPL.

He had time to develop his ideas in connection with the project of the geometry theorem prover, developed under the direction of H. Gelernter at IBM. Also participating were G. Gerbrich and J. Hanson.

Due to the proposal of IBM to build up the "New England Computation Center" for the universities and colleges in this field, McCarthy concentrated on the IBM-704. He developed in Dartmouth the basic functions for a list processing system embedded in FORTRAN.

It is very well known that the word structure of the IBM-704 (the same as of IBM-7090) gave the names for LISP's basic functions. 1957 there were 4 of them : CPR (prefix), CDR (decrement), CTR (tag) and CAR (address part). The function CWR augmented this set in giving the contents of a word. In the begining, all functions had to be used in connection with CWR to give the same result as what we do today with CAR etc. Then an operator CONS of four arguments was invented which served to fill a word with the 4 parts.

The use of the functions came to show a better definition of the basic functions, which allowed CWR to be dropped, and also showed the practical uselessness of the prefix and tag parts (only 3-bit quantities). Gelernter and Gerberich made CONS into a function having the address of the filled word as a result.

A clear shadow of this can be studied in the JACM-paper on the geometry theorem prover (11). Later the team developed FLPL independently of J. McCarthy. The exact dates for this cooperation are not known. The results of the geometry prover project of IBM were published in (12, 13,

14), the first proof could be stated in spring of 1959. During 1958 McCarthy had a subsidy of the A. Sloan foundation and he spent the year at MIT Cambridge thinking about his language and about paper-programming a chess programm in FORTRAN. In doing this he became aware that the means of formulating the expression of conditionnal actions were very poor. It is well known that FORTRAN has only the so called "arithmetical" IF, which allows a test against 0 and 3 jumps to different statement numbers and the "logical" IF-statement, allowing a test and a statement performed if the test is true. But FORTRAN has both forms of IF as statements. (Later the IF-THEN-ELSE came, but it is a statement too). In this chess program McCarthy found it very useful to have a conditionnal expression. He defined a function IF with 3 arguments to do the work, but it is clear that all arguments are computed in FORTRAN and the selection could be done only afterwards. Here was a restriction in FORTRAN and McCarthy thought about it. The result of this was a paper send to Communications of ACM and a lot of propaganda for conditional expressions to come into ALGOL. But the idea was too new and the Communications published the paper in the form of a letter to the editor (15) and the ALGOL committee did not use the proposal.

During the summer of 1958 McCarthy spent some time at IBM (it is not clear in which connection to the FPLP-group) and worked to write programs for symbolic differentiation. Doing this he was forced to write recursive programs and this concept was added to the conditionnal expressions. Another concept which was used in the (paper-) differentiation program was the use of functional arguments. They were very useful for differentiating sums with 3 and more summonds. For writing functional argument McCarthy adopted the way of the Lambda-Calculus (16). (He wrote me that he didn't read the Church paper completely and didn't understand it fully but I guess this was more out of humility).

In this way a lot of basic concept for a language beyond FORTRAN arose and it was very nice for McCarthy to find a medium to discuss and formulate it's final version : in the autumn of 1958 the MIT AI-project was founded.

McCarthy became Assistant Professor of Communication Science (in the Electrical Engineering Department) and Minsky became Assistant Professor (in the Mathematics Department). "The project was supported by the MIT Research Laboratory of Electronics (RLE) ... No written proposal was ever made. When J .Wiesner (the director of RLE) asked Minsky and me what we needed for the project, we asked for a room, two programmers, a secretary and a keypunch; he also asked us to undertake the supervision of some of six mathematics graduate students that RLE had undertaken to support ... " (1)

N. Rochester from IBM was at this time visiting at MIT and participated in the work, and C.E. Shannon was strongly connected with the group. The students were P.W. Abrahams, D.G. Bobrow, K.R. Brayton, L. Hodes, L. Kleinrock, D.C. Luckham and K. Maling ; the programmers were S.R. Russell and D.J. Edwards (the latest is mentioned only after May 1959).

During September and October McCarthy wrote some memos about the new language. At the same time he worked on the advice taker proposal (17). Then he started to write a paper concerning the formalism of conditional expressions ans its impact for recursive function theory. Doing this he developed the universal function Apply. At the same time the students and programmers had to hand-translate some of the older paper-programs of J. McCarthy and some new ones (for reading and printing symbolic

expressions). Doing this the conventions for subroutine entry and exit, for work with the stack and for storage management, were developed. We must remember that none of the members of the AI group had a deeper knowledge of compilers or language implementation!

By writing the read and print programs the syntax of S-expressions was fixed. The available IBM026 key punch has only 47 characters and because of this way the syntax was rather poor. For organizing the work with recursive functions, at the beginning only the IPL way of doing this was known. McCarthy proposed to use, instead of the pushdown lists (locally to each function), linear stacks. This was abandoned soon in favor of a public stack. The work of subroutines was organised in such a way that, at entry time, they saved their local registers in the stack to do the work, restoring them at the end. It is clear that this work was completely new at this time (remember that Dijkstra's paper (18) on implementation of recursive ALGOL using a stack was written in 1960!). More complicated was the problem of managing the list storage. The IPL solution (let the programmer do the work) seemed not to be the goal and the counter-method developed at the same time by Collins (19) was not very well applicable. (The free places are only 6 bits and these are not directly linked together). The storage was large and the problems small and so they shifted the storage manipulation to later times. In transfering the paper programs to assembly programs as parts of the growing program system Steve Russell became soon very experienced. One nice day he saw the APPLY-EVAL-functions on McCarthy's desk and he asked if he should transfer them too. But McCarthy said that this is only theory and not practice. Russel didn't bother about it, took the functions and added them to the system.

This important event most likely happened in November of 1958, then D. Park, who joined the group at this time, claims he would come if these functions were running.

By April 1959 the status was as following (20):

(a)The source language has been developed and is described in several memos from the AI group.

(b)20 useful subroutines have been programmed in LISP, hand-translated into SAP and checked out on the IBM 704. These include routines for reading and printing list structures.

(c)A routine for differentiating elementary functions has been written. A simple version has been checked out etc...

(d)A universal function APPLY has been written in LISP, hand-translated, and checked out. Given a symbolic expression for a LISP function and a list of arguments, APPLY computes the result of applying the functions to the arguments. It can serve as an interpreter for the system and is being used to check out programs in the LISP language before translating them to machine language.

(e)Work on a compiler has been started. A draft version has been written in LISP and is being discussed before it is translated into machine language or checked out with APPLY...

LISP was used at this time for calculations of properties of linear passive networks (Rochester, Goldberg, Rubenstein, Edwards, Markstein). It may be that some of the work for this was written in FLPL. Further use was for chess (McCarthy, Shannon, Kleinrock and Abrahams). "Other projects include the Advice Taker, visual pattern recognition and an

artificial hand. Work has been started by Bobrow, Maling and Park on a proof checker for predicate calculus and by Slagle on a program for computing indefinite integrals".

The same source (20) contains the first version of the well known Communications of ACM paper (21).

c) TOWARDS LISP 1

As we can see, in April of 1959 the author himself didn't understand the interpreter as the main instrument for making the language run. The S-language was only used for data.

But it is clear: the possibility for fast and direct access to the machine had its consequences. So over the time all the students and the programmers used more and more the S-language directly for programming. The M-language became more a means for communicating about LISP.

"The unexpected appearance of an interpreter tended to freeze the form of the language, and some of the decisions made rather lightheartedly for the "Recursive functions..." paper later proved unfortunate. These included the COND notation for conditional expressions which leads to an unnecessary depth of parentheses, and the use of the number zero to denote the empty list NIL and the truth value false. Besides encouraging pornographic programming, giving a special interpretation to the address 0 has caused difficulties in all subsequent implementations.

Another reason for the initial acceptance of awkwardnesses in the initial form of LISP is that we still expected to switch to writing programs as M-expressions. The project of defining M-expressions precisely and compiling them or at least translating them into S-expressions was neither finalized nor explicitly abandonned. It just receded into the indefinite future, and a new generation of programmers appeared who preferred internal notation to any FORTRAN-like or ALGOL-like notation that could be devised."(1)

It is not very clear if the M-language at the beginning (t.m. 1958) has included the PROG feature. It is very probably that the former programs were all more or less FORTRAN, so sequential parts were usual. The capacity to translate into S-expressions should be made the introduction of a special notation in the M-language (But this is a hypothesis of mine and can be proven only by the old memos. At present nobody, incuding McCarthy can find them).

The growing system, in any case the work of Goldberg in formula manipulation (symbolic matrices!) was one nice day full. So there was a need for a solution. The idea was to reclaim all used data and form a list of free registers. McCarthy developped the classical algorithm that uses a stack for saving unmarked list parts. Very probably this was not recursive in spite of the external notation and discussion. It is very curious to see, besides its stack usage, the nearly optimal performance of this version of GC-algorithm. It was only Toshiaki Kurokawa who found a faster solution (22). The work of Goldberg came to an end in the summer of 1959 (23). J. Moses claims his

simplificator program as the first ever written and he says it should be written in assembly language (24). The next problem solved in 1959 is the problem of free variables. All LISP-people know it as the FUNARG-problem.

J. Slagle, writing his program for integration, tried to write a function which could apply a test-predicate to all elements of an S-expression. It is very near to the function TESTR as described by R.A. Saunders (25).-See for a complete version in (1).

The function didn't run correctly and nobody had any idea why. The programmers and Slagle went to McCarthy...."..naturally he complained. And I never understood the complaint very well, namely, I said : 'oh, there must be a bug somewhere, fix it!' And Steve Russell and Dan Edwards said, there is a fundamental difficulty and I said, there can't be, fix it, and eventually they fixed it and there was something they called the funarg device. He tried to explain it to me but I was distracted or something and I didn't pay much attention so I didn't really learn what the funarg thing did until really quite a few years later. But then it turned out that these same kinds of difficulties appeared in ALGOL and at the time, the LISP solution to them, the one that Steve Russell and Dan Edwards had simply cooked up in order to fix this bug, was a more comprehensive solution to the problem than any which was at that time in the ALGOL compiler."(2)

D. Park claims that Patrick Fischer should have a part in this solution (1).

The next steps towards the first complete LISP-systems are :

- introduction of property lists (and the usage of properties EXPR etc.)

- introduction of pseudofunctions RPLACA and RPLACD

- introduction of floating point numbers to make arithmetic possible without using lists. A very impressive picture of how the early LISP did work with numbers can be studied using the paper of Jenkins and Woodward (26). The floating point numbers in LISP 1 were to be quoted if they came into the interpreter. Three functions (SUM, PRDCT and EXPT) could be used for arithmetical work.

The state of the LISP-language and system in september of 1959 could be studied if we had the paper (27). In November, 59, McCarthy started to write the first manual. But he didn't finish it and the LISP 1 manual was written by Dr. Phyllis Fox, who had joined the AI group in September, 1959.

In January 1960 the compiler, written by R. Brayton under assistance of D. Park was running. It is not very clear why McCarthy remembers only a failing compiler project in connection with LISP 1. Maybe he was waiting for an M-language compiler and they constructed an S-language compiler. The compiler was, if we follow the reports (28,29) indeed running.

This is also the claim of D. Park. The report (28) gives speed-up figures of 30 minutes in interpretative mode to 0.6 minutes compiled. The result was to some degree very similar to our later compilers, and to another degree very different. First, the compiler produced a LAP-code list. Then it was added as pseudo-function to the system. Many parts of it may be the model of the later compilers (30). The code

itself is very different in its kind to the later delivered code. For local variables it generated GENSYMs which were appended to the end of the code. Recursive functions had to shift their local variables in the stack before they started. Labels were generated as instructions having the first place used (all instructions in LAP format had a first symbol field). CONS, NULL, PROG, RETURN, GO, CAR, CDR and possibly ATOM were translated open.

The well known paper (21) tells not very much about the state in March 1960. Only error diagnostics and tracing facilities are mentioned.

The manual (29) shows an interesting system. We found ca. 90 functions some of which were very singular (like the function CP1 in LISP 1.5), others stem from Goldberg's work (SIMPLFY, DIFF, MATRIXMULTIPLY, REDUCE and REDUCETONXN for simplification, differentiation, matrix multiplication, matrix reduction resp.) or from former assembled programs.

An important (for the time!) part was the "Flexowriter system" which was constructed by Luckham and Edwards. Using a pre time-sharing possibility "time stealing", the user could communicate directly with the LISP system (cf. the story told by McCarthy in (1)).

d) TOWARDS LISP 1.5

We have not very much information about people and the work done in the time from 1960 to 1962. What we know is, that in March 1960 the LISP implementation for IBM709 was started. Then we know the start and the end product - LISP 1 and LISP 1.5 respectively - and that some new people are involved now : T.P. Hart, M.I. Levin, B. Raphael were new. But Luckham and Park don't work very much with LISP.

From the old students only Bobrow, Slagle and Abrahams wrote a thesis using LISP. Park writes about this : "... I think that McCarthy, while he was ahead of all of us in seeing the significance of the thing, and in his research ambitions, was as surprised as anyone that something of such general significance to computer people had emerged. LISP made it all so easy. Paradoxically, for those of us looking for research topics, it make look things too easy, in some sense. McCarthy went on to capture a good deal of the theoretical importance in his "Mathematical Thery of Computation" work - but he pursued this almost entirely on his own.

The message caught on much more slowly for the rest of us, or at least for me. None of the research students listed as authors of LISP 1 went on to do doctorate work using computers or about programming. For myself, having been excited by LISP, and having listened with ill-judged scorn to McCarthy's initial work on his theory of computation, which seemed too straightforward for intellectual ambitious students like me to worry with, I ended up writing a thesis on mathematical logic ..."(31).

At the end of this 2-years period, Rochester is back to IBM, but H. Rogers Jr. and H. Teager had joined the group. Further persons are : D.A. Dawson, E.L. Ivie, P.G. Jenson and U. Shimony.

The new system was running better and better and in august of 1962 the new manual was completed. It seemed to describe not the old language/system but a better thing. The plans for the second variant were discussed at length at that time. So it seemed not quite correct to call the system LISP 2 - a medium name was searched for and LISP 1.5 used.

The differences between LISP 1 and LISP 1.5 are :

1. LISP 1.5 allows integers, has other internal representation for floating point numbers and allows both to be evaluated without quoting.

2. The set of arithmetic functions is completely different and remarkably larger.

3. LISP 1.5 allows arrays.

4. The user has to deliver doublets in LISP 1.5 : the system starts always with an empty alist. (In LISP 1 there were triplets of function, argumentlist and alist. Top-level : APPLY). Toplevel now : EVALQUOTE.

5. LISP 1.5 allows pointed pairs.

6. At the same time the stucture of the alist and the arguments for SASSOC are altered.

7. With the exception of LIST, in LISP 1 no function could have an indefinite number of arguments. Now a serie of such functions exists.

8. LISP 1 did allow only LISP input. LISP 1.5 has now a large set of I/O functions.

9. The flexowriter system isn't in existence. Device-I/O is handled by a monitor.

10. LISP 1.5 introduced the $$x...x notation for unusual atoms.

11 In LISP 1 there were difficulties with multiple CAR-CDR's. Some functions work for this (DESC, MAKCBLR, PICK). In LISP 1.5 all is solved by shifting the problem to the user.

12. LISP 1.5 uses TRACE instead TRACKLIST.

13. The compiler is completely new.

14. A new LAP.

15. LISP 1.5 introduces CSET and CSETQ for setting constants (both are EXPRs !). LISP 1 has neither and has two distinct indicators for constants (APVAL and APVAL1).

16. LISP 1.5 alters the names : what is in LISP 1 the property-list is in LISP 1.5 the association list. What was the association list is now the P-list.

17. A lot of internal changes, some of them visible for the user.

18. LISP 1.5 introduces the error function ERRORSET.

Very soon the LISP system is adopted inside the time-sharing system. We drop here to mention the work of McCarthy concerning time-sharing. In summer 1962 McCarthy moves to Stanford. The further developement is characterized by the strong connection to CTSS at MIT and the work of McCarthy to build up similar facilities at Stanford University. The delivery of a PDP-6 in December 1964 to MIT is the start point for LISP 1.6 that later (1967) is coming to Stanford. The concrete history between 1962 and 1966 is not full clear.

After 1966 we have for MACLISP J.L. White's paper (32).

The history of INTERLISP is clear in outline (as everybody can see in the preface of the manual (33)), but the connection to BBN's former work in LISP (as reported by Berkeley (34)) is unknow. Most of LISP people know the Berkeley-Bobrow book (35) about LISP. Bobrow has planned two further books (The Nature, Use and Implementation of the Computer Language LISP, Cambridge 1967; and LISP Applications, 1970) but neither seems to be appeared. Very unclear is the history of UNIVAC-LISP. We have only the author (Eric Norman), the location : University of Wisconsin,and a guessed time : 1969.

The first international echo on the LISP developement came from G.B. Here in London a group around C. Strachey registrated the paper (21) and during a tea-discussion Mike Woodger proposed D. Jenkins from the Royal Radar Establishment at Malvern to implement LISP. This was done by Jenkins and Woodward in 1960-1961. They report in their paper very little about the system on TREAC and more about LISP itself (36).

The next LISP implementation was done by H. McIntosh at the University of Florida in 1962. Very short after that L. Hawkinson wrote a LISP-system for the IBM709 at Yale University. Both of them went then to Mexico City and combined their work in LISP. In december 1963 to January 1964 the 1st International LISP Conference was held in Mexico City. Very little is known about; we have only a paper of M. Minsky concerning garbage collection and a paper of D. Edwards concerning secondary storage.

To come to an end, we mention only the beginning in some other coutries : in 1965 J. Cohen implements LISP in ALGOL (38). In 1966 J. Kent implements LISP on a CDC3600 in Oslo, Norway. 1967 he implements with the help of J. Bolce LISP on IBM/360 in Canada. This work was concluded at Stanford in 1967. In 1966 V. D. Poel and V. D. Mey started in the Netherlands using a PDP-8. Delft is now a location where a lot of different implementations were made (mostly on PDP and X8). Actual work with LISP in Sweden seems to be started 1968 after foundation of the Datalogilaboratoret under the direction of E. Sandewall by transfering and upgrading the CDC3600-system of Kent.

After the IFIP-symposium on symbol manipulation (39) some new LISP activities came out. In Poland S. Waligorski embedded LISP in ALGOL on a GIER-computer. 1968 Lawrow started in Moskow with some help from Berkeley on the BESM-6. In the same year we have the letter of K. Bahr to the CACM (40) indicating, that LISP is in Germany. In Eastern Germany we started 1970. In Czechoslowakia they started 1971. The same holds for Hungaria. In Italy a lot of different LISP systems seem to be

imported. Some work is known concerning MAGMA-LISP (41).

I am writing a book about LISP, which will contain not very much about programming, a chapter on application fields, a chapter on basic ideas, a chapter on history, a chapter on comparison of "famous" systems and a last chapter on LISP implementation. For the historical chapter every help will be recommanded.

REFERENCES

(1) J. McCarthy: LISP History. SIGPLAN Symposium on History of Programming Languages, Los Angeles, June 1978.

(2) J. McCarthy: Talk about LISP history, held in Summer 1974 at MIT.

(3) J. McCarthy, R. Brayton, D. Edwards, P. Fox, L. Hodes, D. Luckham, K. Maling, D. Park, S. Russel: LISP 1 Programmer's Manual, March 1 1960, MIT, Comp. Center and RLE, Cambridge, Mass.

(4) J. McCarthy: Inversion of Turing Machines, in J. McCarthy and C. Shannon (Eds.): Automata Studies, Princeton, 1956.

(5) P. McCorduck: History of AI, Advance Papers of 5th IJCAI, 1977

(6) A. Newell, H. Simon: The Logic Theory Machine - A Complex Information System, IRE Trans. Information Theory, Vol IT-2, No. 3, Sept. 1956, pp 61-79.

(7) Preliminary Report: Specification for the IBM Mathematical Formula TRANslating System, FORTRAN, IBM Corp., Prog. Res. Group., Appl. Sci. Div., 1954.

(8) The FORTRAN Automatic Coding System for the IBM 704 EDPM, Programmer's Manual, IBM Corp.,32-7026, Oct. 1956.

(9) A. Newell, J.C. Shaw, H. Simon: Programming the Logic Theory Machine, Proc. WJCC, Feb. 1957.

(10) A. Newell, F.M. Tonge: An Introduction to IPL V, Comm. ACM 3,4,1960.

(11) H. Gerlernter, J.R. Hansen, C.L. Gerberich: A FORTRAN Compiled List Processing Language, Journ. ACM 7,2, 1960.

(12) H. Gerlernter, N. Rochester: Realization of a Geometry Theorem Proving Machine, ICOI Proc., Paris 1959.

(13) H. Gerlernter, J.R. Hansen, D. Loveland: Empirical Explorations of the Geometry Theorem Proving Machine, Proc WJCC 1960.

(14) J.R. Hansen: The Use of the FORTRAN Compiled List-Processing Language, IBM Th. Watson Res. Center, Res. Rpt.:RC-282, June 1960.

(15) J. McCarthy: Letter to the Editor, Comm. ACM 2,8, 1959.

(16) A. Church: The Calculi of Lambda-Conversion, Princeton, 1941.

(17) J. McCarthy: Programs with Common-Sense, Proc. of Symp. on Mechan. of Thought Processes, Nat. Phys. Lab., Teddington GB, Nov. 1958.

(18) E. Dijkstra: Making an ALGOL translator for the X1, reprinted in: Ann. Rev. Automatic Programming, R. Goodman (Ed.), Pergamon Press, 1963.

(19) G.E. Collins: A Method for Overlapping and Erasure of Lists, Comm. ACM 3,12,1960.

(20) Quaterly Progress Report no 53, april 1959, RLE, MIT, Cambridge, pp 122-152.

(21) J. McCarthy, Recursive Functions of Symbolic Expressions and their Computation by Machine, Comm ACM, 3, 3, 1960.

(22) Toshiaki Kurosowa, A New Save and Fast GC-algorithm, Tokyo, 1976.

(23) S. H. Goldberg, Solution of an Electrical Network Using a Digital Computer, MIT, MS Thesis, 1959.

(24) J. Moses, Simplification - a Guide to the Perplexed, 2nd Symp. on alg. and symb. manipulation, ACM, 1971.

(25) R. A. Saunders, LISP - on the Programming System, in (34).

(26) D. R. Jenkins, Woodward, Atoms and Lists, Comp.J., 4 april 1961.

(27) J. McCarthy, LISP - a Programming System for Manipulating Symbolic Expressions, Annual Meeting of ACM, MIT, Cambridge, sept. 2-4, 1959.

(28) Quarterly Progress Report no 56, january 15, 1960, RLE, MIT, Cambridge, pp 158-161.

(29) see (3).

(30) R. A. Saunders, The LISP Listing for the Q-32-Compiler, in (34).

(31) D. Park, personal communication, 1978.

(32) J. White, LISP : Program is Data - a Historical Perspective on MACLISP, MACSYMA 1977 users conference, Berkeley.

(33) W. Teitelman, INTERLISP Reference Manual, Palo Alto, 1974.

(34) E. C. Berkeley, D. G. Bobrow (eds), The Programming Language LISP, its Operation and Applications, Cambridge, 1964.

(35) E. C. Berkeley, LISP - an Introduction, Computers and Automation, Sept 1964.

(36) see (26).

(37) M. Minsky, A LISP Garbage Collector Algorithm Using Serial Secondary Storage, AI-memo 58, MAC-M-129, MIT, 1963.

(38) D. J. Edwards, Secondary Storage in LISP, AI-memo 63, MAC-M-128, MIT, dec. 27, 1963.

(39) D. G. Bobrow (ed), Proc IFIP Conf. on Symbol Manip. Lang., Pisa, sept 1966, N.Y. 1968.

(40) K. Bahr, Letter to the editor, Comm.ACM, 11, 6, 1968.

(41) Montangero, Pacini, Turini, MAGMA-LISP, Adv. Papers 4th IJCAI 1974.